CSCI-1680
Transport Layer I

Nick DeMarinis

Based partly on lecture notes by Rodrigo Fonseca, David Mazières, Phil Levis, John Jannotti
• IP: due tonight!
  – Look for email today/tomorrow about grading meetings
    + feedback survey

“Between the time you’ve handed in and the demo meeting, you can continue to making small changes and bug fixes and push them to your git repo”
Administrivia

• IP: due tonight!
  – Look for email today/tomorrow about grading meetings + feedback survey

“Between the time you’ve handed in and the demo meeting, you can continue to making small changes and bug fixes and push them to your git repo”
  – OK: Fixing bugs, code cleanup, README
  – Not OK: Implementing RIP, adding new features
• HW2 is out (finally!):  Due Monday, Oct 30
• HW3 will be super short:  out Oct 31, due Nov 7

• TCP:  Should be out tomorrow
  – Gearup on Monday, Oct 23 6-8pm in CIT316
Today

Light overview of the transport layer and TCP

– Why we need TCP
– What components are involved
– What you will do in the project
Transport layer: the story so far

- Provides support for different applications via ports
- OS provides interface to applications via sockets
Transport layer: the story so far

- Provides support for different applications via ports
- OS provides interface to applications via sockets

⇒ For now: transport layer is part of OS, service provided to apps
The headers

**UDP**

- Source Port [0-15]
- Destination Port [16-31]
- UDP Length
- UDP Checksum
- Data

**TCP**

- Source Port [0-15]
- Destination Port [16-31]
- Sequence Number
- Acknowledgement Number
- Data Offset
- Reserved
- Urgent Pointer
- Window Size
- Checksum
- Options
- Data
Port numbers are part of these headers
=> OS uses these to map to sockets
Motivation: sending a big file

A problem, in pseudocode:

```go
func sender() {
    fd, _ := os.Open("all-my-files.zip")
    conn, _ := net.Dial("1.2.3.4:80")
    buf := ReadTheWholeFile(fd)
    conn.Write(buf)
}
```

```go
func receiver() {
    conn, err := net.Listen("80")
    buf := make([]byte, . . .)
    conn.Read(buf)
    fd = os.Open("copy-of-files.zip")
    fd.Write(buf)
}
```

What are some challenges with implementing the network part?
Motivation: sending a big file

A problem, in pseudocode:

```go
func sender() {
    fd, _ := os.Open("all-my-files.zip")
    conn, _ := net.Dial("1.2.3.4:80")
    buf := ReadTheWholeFile(fd)
    conn.Write(buf)
}
```

```go
func receiver() {
    conn, err := net.Listen(":80")
    buf := make([]byte, . . .)
    conn.Read(buf)
    fd = os.Open("copy-of-files.zip")
    fd.Write(buf)
}
```

⇒ How do we get data from A->B, reliably?
How does the transport layer help us do this?
UDP: User Datagram Protocol
Send a message between ports… and nothing else
UDP: What could possibly go wrong?

Map of the Internet, 2021 (via BGP)
OPTE project
Problem: Reliability

Packets could...

• Dropped packets
• Duplicate packets
• Packets arrive out of order

Multiple hops and paths => Lots of opportunities for failure!
=> TCP has mechanisms to deal with this
Also: performance challenges

- Hosts have different (and unknown!) resources
- Network has unknown resources
  - Varying RTT, link bandwidth
So how does it work?
TCP: the big picture
TCP: THE BIG PICTURE:

Connect
Send

APP
TCP STACK (KERNEL)

TCP - THE CONNECTION STATE:

Send buffer: data waiting to be sent out

Receive buffer

Sending side:
- Buffers data from app to be sent
- Divides data into segments
- Track which segments have been received, which have been dropped (retransmit on timeout)
- Flow control: if receiver has no more buffer space, stop sending

Receiving side:
- Arrange segments in order in recv buffer
- Sends back “acknowledgements” + other info
- App “pulls” data from the receive buffer, frees up more space for data to arrive from network

In practice, both sides of the connection can send and receive (full-duplex)
=> Both sides have send and receive buffers
=> (Can use the same socket to send and receive)
TCP: Key features

• Initially: RFC 793 (1981) (+ many others now)

• Creates concept of connections between two endpoints
  => Each connection has its own state

A ——— B
TCP STATE

EACH CLIENT-SERVER CONNECTION HAS ITS OWN TCP STATE.
TCP: Key features

• Initially: RFC 793 (1981) (+ many others now)

• Creates concept of connections between two endpoints
  => Each connection has its own state

• End-to-end protocol
  – Minimal assumptions on the network
  – All mechanisms run on the end points (i.e., not routers)
TCP Header

- Source Port
- Destination Port
- Sequence Number
- Acknowledgement Number
- Window Size
- Checksum
- Urgent Pointer
- Options
- Data
Important Header Fields

• **Ports: multiplexing**
• **Sequence number**
  – Where segment is in the stream (in **bytes**)
• **Acknowledgment Number**
  – Next expected sequence number
• **Window**
  – How much data you’re willing to receive
• **Flags**...
Important Header Fields: Flags

- **SYN**: establishes connection ("synchronize")
- **ACK**: this segment ACKs some data (all packets except first)
- **FIN**: close connection (gracefully)
- **RST**: reset connection (used for errors)
- **PSH**: push data to the application immediately
- **URG**: whether there is urgent data
Less important header fields

- **Checksum**: Very weak, like IP
  - Has weird semantics ("pseudo header"), more on this later…

- **Data Offset**: used to indicate TCP options

- **Urgent Pointer**: Unused (out of scope for this class)
### RFC documents

* RFC 793 – TCP v4
* RFC 1122 – includes some error corrections for TCP
* RFC 1323 – TCP Extensions for High Performance [Obsoleted by RFC 7323]
* RFC 1948 – Defending Against Sequence Number Attacks
* RFC 2018 – TCP Selective Acknowledgment Options
* RFC 5681 – TCP Congestion Control
* RFC 6247 – Moving the Undeployed TCP Extensions RFC 1072, 1106, 1110, 1145, 1146, 137
* RFC 6298 – Computing TCP’s Retransmission Timer
* RFC 6824 – TCP Extensions for Multipath Operation with Multiple Addresses
* RFC 7323 – TCP Extensions for High Performance
* RFC 7414 – A Roadmap for TCP Specification Documents
* RFC 9293 – Transmission Control Protocol (TCP)

There must be a better way!!
RFC9293

The One RFC

STD: 7
Request for Comments: 9293
Obsoletes: 793, 879, 2873, 6093, 6429, 6528, 6691
Updates: 1011, 1122, 5961
Category: Standards Track
ISSN: 2070-1721

Internet Standard

W. Eddy, Ed.
MTI Systems
August 2022
Establishing a Connection

Goals
• Contact the other side (or error)
• Both sides agree on initial sequence numbers
Establishing a TCP Connection

1. Sender sends SYN with random sequence number X
2. Receiver sends SYN+ACK with its own random sequence number Y, acknowledges sender’s sequence number with ACK=X+1
3. Sender acknowledges receiver’s sequence num with ACK for Y+1 (packet also has SEQ=X+1, since it comes after packet (1))

⇒ 3-way handshake
TCP State Diagram

- **TCP State Diagram**
  - **Setup**
  - **Normal Operation**
  - **Connection Teardown**

**States and Transitions**:
- **CLOSED**
  - (Start)
  - LISTEN
  - CONNECT/SYN (Step 1 of the 3-way-handshake)
  - SYN/SEND
  - SYN/SEND/SYN
  - SYN/SEND/SYN+ACK
  - SYN/SEND/SYN+ACK (Simultaneous Open)
  - ESTABLISHED
  - SYN+ACK
  - SYN+ACK+ACK
  - FIN+ACK
  - FIN+ACK+ACK
  - PASSIVE CLOSE
  - CLOSE WAIT
  - CLOSE FIN
  - LAST ACK
  - TIMEOUT
  - (Go back to start)

**Actions**:
- ACK
- FIN
- RST
- SET

**Paths**:
- **Listen**: client->server
- **Normal**: server->client
- **Teardown**: client->server

**Special Notes**:
- Unusual event: client上诉 path
- Server->sender path

**Key Phases**:
- **Setup**: SYN/SEND
- **Normal Operation**: SYN/SEND/SYN
- **Teardown**: FIN/FIN

**Phase Diagrams**:
- **FIN WAIT 1**: ACK
- **FIN WAIT 2**: FIN
- **CLOSING**: ACK
- **TIME WAIT**: Timeout
- **CLOSED**: (Go back to start)
EXTRA STUFF

FOR NEXT CLASS

→
Sequence numbers

How to pick the initial sequence number?

• Protocols based on relative sequence numbers based on starting value
• Why not start at 0?

• RFC9293, Sec 3.4.1: Procedure for picking ISN, based on timer and cryptographic hash
  => For project, just pick a random integer :)

•
How do we tell two connections apart?

=> Port numbers

– 5-tuple (proto., source IP, source port, dest IP, dest port) => 1 Connection

– Kernel maintains socket table: maps (5-tuple) => Socket

• If a 5-tuple is reused => new ISN, so sequence numbers likely out of range from past connection
<table>
<thead>
<tr>
<th>Proto</th>
<th>Recv-Q</th>
<th>Send-Q</th>
<th>Local Address</th>
<th>Foreign Address</th>
<th>(state)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>10.3.146.161.51094</td>
<td>104.16.248.249.443</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>10.3.146.161.51076</td>
<td>172.66.43.67.443</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp6</td>
<td>0</td>
<td>0</td>
<td>2620:6e:6000:900.51074</td>
<td>2606:4700:3108::443</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>10.3.146.161.51065</td>
<td>35.82.230.35.443</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>10.3.146.161.51055</td>
<td>162.159.136.234.443</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>10.3.146.161.51038</td>
<td>17.57.147.5.5223</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp6</td>
<td>0</td>
<td>0</td>
<td>*.*51036</td>
<td><em>.</em></td>
<td>LISTEN</td>
</tr>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>*.*51036</td>
<td><em>.</em></td>
<td>LISTEN</td>
</tr>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>127.0.0.1.14500</td>
<td><em>.</em></td>
<td>LISTEN</td>
</tr>
</tbody>
</table>
Keeping state: the TCB

State for a TCP connection kept in Transmission Control Buffer (TCB)

- Keeps initial sequence numbers, connection state, send/recv buffers, status of unACK’d segments, ...
- When to allocate?
Keeping state: the TCB

State for a TCP connection kept in Transmission Control Buffer (TCB):

- Keeps initial sequence numbers, connection state, send/recv buffers, status of unACK’ed segments, ...

- When to allocate?
  - Server: listening on a connection*
  - Client: Initiating a connection (sending a SYN)
  - Server: accepting a new connection (receiving SYN)
Recall: the socket table

- Each connection has an associated TCB in the kernel
- For each packet, kernel maps the 5-tuple (tcp/udp, local IP, local port, remote IP, remote port) => socket
- Depending on socket type, socket contains TCB
Two "types" of sockets:

- "Normal" sockets
- Listen sockets
“Normal” sockets

- Connection between two specific endpoints
- Can send/recv data

Listen sockets

- Created by receiver to accept new connections
- When a client connects, client info gets queued by kernel
- When server process calls accept(), **a new (“normal”) socket is created between the server and that client**
NOTA BENE: This diagram is only a summary and must not be taken as the total specification. Many details are not included.
NOTA BENE: This diagram is only a summary and must not be taken as the total specification. Many details are not included.

active OPEN
\      \       \       \
create TCB \       create TCB
|      |       V
CLOSED |

CLOSE
\      \       \       \       \       \       \       \       
send SYN

rcv RST (note 1)
\       \       \       \       \       \       \       \       
rcv SYN

rcv SYN
\       \       \       \       \       \       \       \       
snd SYN

snd SYN,ACK
\       \       \       \       \       \       \       \       
send SYN,ACK

SYN

RCVD

snd SYN,ACK

rcv ACK of SYN
\       \       \       x
V       \       V

CLOSE
\       \       \       \       \       \       \       \       
snd FIN

rcv SYN
\       \       \       \       \       \       \       
send SYN,ACK

LISTEN
\       \       \       \       \       \       \       
send SYN

SEND
\       \       \       \       \       \       \       \       
snd ACK

SYN SENT

ESTAB
SYN flooding

What happens if you send a huge number of SYN packets?
A hacky solution: SYN cookies

- Don’t allocate TCB on first SYN
- Encode some state inside the initial sequence number that goes back to the client (in the SYN+ACK)
A hacky solution: SYN cookies

- Don’t allocate TCB on first SYN.
- Encode some state inside the initial sequence number that goes back to the client (in the SYN+ACK).
- What gets encoded?
  - Coarse timestamp
  - Hash of connection IP/port
  - Other stuff (implementation dependent)
- Better ideas?
Next class

• Sending data over TCP
• FIN bit says no more data to send
  – Caused by close or shutdown
  – Both sides must send FIN to close a connection

• Typical close
The IPv4 Header

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bit</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td></td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>Version</td>
<td>IHL</td>
<td>TOS</td>
<td>Total length</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Identification</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Flags</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Fragment offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TTL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Protocol</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Header checksum</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Destination address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Options</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 20 bytes
- 0-40 bytes
- Up to 65536 bytes
# The IPv4 Header

Defined by RFC 791

**RFC (Request for Comment):** defines network standard

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-4</td>
<td>Version</td>
<td>version number</td>
</tr>
<tr>
<td>4</td>
<td>IHL</td>
<td>fragment offset</td>
</tr>
<tr>
<td>4-8</td>
<td>TOS</td>
<td>type of service</td>
</tr>
<tr>
<td>8</td>
<td>Total length</td>
<td>total length</td>
</tr>
<tr>
<td>9-16</td>
<td>Identification</td>
<td>source address, destination address</td>
</tr>
<tr>
<td></td>
<td>Flags</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fragment offset</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>TTL</td>
<td>time to live</td>
</tr>
<tr>
<td>16-20</td>
<td>Protocol</td>
<td>protocol number</td>
</tr>
<tr>
<td>20-31</td>
<td>Header checksum</td>
<td></td>
</tr>
<tr>
<td>32-40</td>
<td>Source address</td>
<td></td>
</tr>
<tr>
<td>41-60</td>
<td>Destination address</td>
<td></td>
</tr>
<tr>
<td>61-64</td>
<td>Options</td>
<td></td>
</tr>
<tr>
<td>65-6536</td>
<td>Data</td>
<td></td>
</tr>
<tr>
<td>6537-65535</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Description</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 bytes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0-40 bytes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Up to 65536</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>